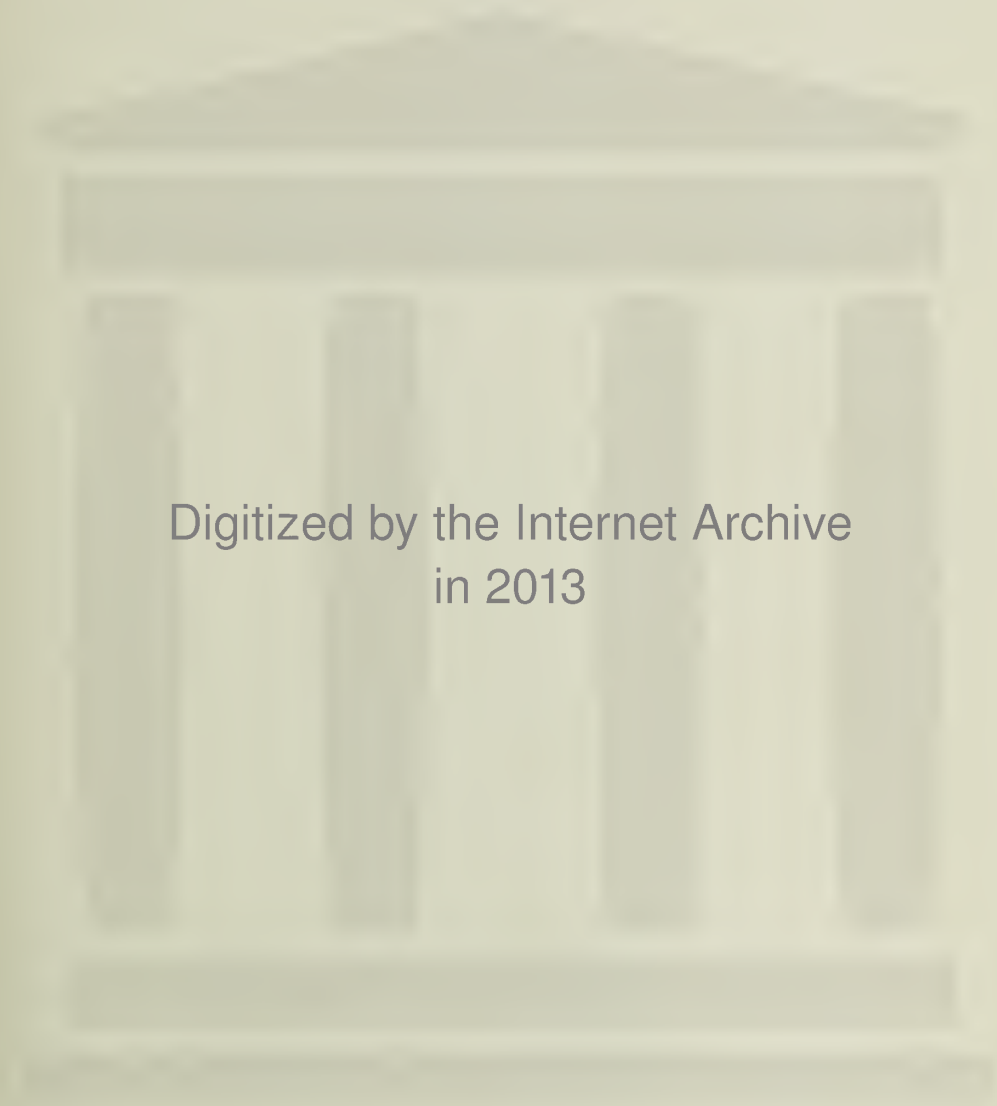


UIUCDCS-R-77-846

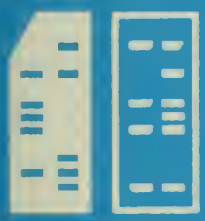
UILU-ENG 77 1704

The VL Relational Data Sublanguage for an Inferential Computer Consultant

by

Richard N. Schubert

October, 1977



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

THE VL RELATIONAL DATA SUBLANGUAGE FOR AN INFERENTIAL COMPUTER CONSULTANT

BY

RICHARD NEAL SCHUBERT

B.S., University of Illinois, 1974

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1976

Urbana, Illinois

TABLE OF CONTENTS

1. INTRODUCTION

An Inferential Computer Consultant is being designed and implemented at the University of Illinois by a research group headed by Professor R.S. Michalski. The computer consultant is intended to extend the capabilities of current information systems by including deductive capabilities and introducing inductive capabilities. Induction is performed using Variable-Valued Logic techniques [1] on sets of facts called event sets. These event sets are most naturally stored using relational tables as proposed by Codd[2].

In the Variable-Valued Logic system VL1[3] an event is an ordered list of values of a corresponding list of descriptors. An event set is simply a set of events corresponding to one list of descriptors. It therefore seems natural to represent an event set by a relational table, where a row (or tuple) corresponds to an event and a column (or domain) corresponds to a descriptor. In order to allow for the creation and manipulation of these relational tables, a data base sublanguage (intended to be a subset of the entire language for communication with the computer consultant) has been developed. The description of this sublanguage is the object of this thesis.

The design of the sublanguage was begun by Professor Michalski for a course he taught on information systems at the University of Illinois in the spring of 1976 after he became disenchanted with some of the awkward constructs of Codd's relational data sublanguage ALPHA[4]. The basic design was taken from a combination of ALPHA and the language VL1. This basic design was then organized and extended by the author. A computer program to implement the sublanguage was initiated by Trevor Morgan, Greg Lyons, and Warren Emery as a semester project for the information systems course and is being continued by the author.

2. DESCRIPTION OF THE SUBLANGUAGE

The VL data sublanguage has instructions for creating, retrieving, and modifying relational tables plus some which use existing tables for the purposes of induction or deduction. The instructions which make up each of these categories are explained in detail below.

Relational tables are created and expanded by two instructions, the DEFINE and the ADD. The DEFINE instruction is used to define the characteristics of relational tables. The keyword RT following DEFINE signals that the user is defining relational tables, while the keyword EVENT in that position indicates that the user is defining events, which may be thought of as relational tables with only one row; if no keyword is given (following DEFINE), then RT is assumed by default.

In defining a new table, the user specifies the table name, the names of the descriptors (which are the column headings for the table), and the names of those descriptors (if any) which make up the key of the table. The key is a subset of the descriptors which can be used to identify a given event; the key must be unique from one event to another. Usually there is only one descriptor in the key, although any number of the descriptors may make up the key. These descriptors should be the first given in the

descriptor list (in this definition) and should occur in the same order in the descriptor list as they do in the key list. For example in defining a relational table T with descriptors A, B, C, D, and E, if the key is C,D, then the table would be defined using the instruction:

```
DEFINE T(C,D,A,B,E) KEY:=C,D END
```

This table, as defined above, will be used in examples throughout this chapter. The table name (T, in this case) should not be used anywhere else as a table name, an event name, a descriptor name, or a value name, and must be distinct from the keywords of the sublanguage. The same restrictions hold for the descriptor names except that the same descriptor may appear in more than one table. More than one relational table may be defined in the same DEFINE instruction (which is delimited by the keywords DEFINE and END) simply by listing the definitions one after another (the optional keyword RT occurs only once, after the word DEFINE).

In defining a new event, the name of the event is given followed by the descriptor names and their corresponding values. A value may be given as an integer, a real number, an arithmetic expression of integers and reals, a character string (enclosed in single quotes), or a value name, which is an alphanumeric string which begins with a letter. The

restrictions on the name of the event are the same as those for that of a new relational table. The purpose of defining an event is so that it may later be added to some relational table; however, the relational table need not exist prior to defining the event. Similarly, the descriptors and/or their corresponding values need not exist at this time--using new descriptor (value) names in a DEFINE EVENT instruction establishes them as descriptors (values). More than one event may be defined in one DEFINE instruction by listing definitions one after another (the keyword EVENT occurs only once, after the word DEFINE). A typical instruction might be:

```
DEFINE EVENT
   E1:=  (A:=10,B:=2.3+5.7,D:=TABLE,C:='GOOD DAY')
   E2:=  (B:=73.1,SIZE:=LARGE,WEIGHT:=185)
END
```

The ADD instruction, which is used to add events to relational tables, has three forms. The first, which would usually be used when originally setting up a relational table, offers the capability of adding many events to one table in the same instruction. After specifying the name of the relational table to which the events are to be added, the user is placed in insert mode, which the user ultimately terminates by typing the word END. Prior to typing END, the

user enters the values for each of the descriptors in each event to be added; the values within each event are entered in the same order that the corresponding descriptors were entered in the DEFINE instruction for that relational table (thus the descriptor names need not--and must not--be specified). A typical instruction might be:

```
ADD TO T
    (NOW,CHAIR,6,45.2,42)
    (BAND,DESK,7,36.0,29*37)
    (THING,LAMP,23,29.1,16)
END
```

These three events would be added to the end of T. The second event demonstrates the use of an arithmetic expression ("29*37") as the value of the descriptor E.

The second form of the ADD instruction is similar to the first except that only one event is listed and it is entered before the name of the table to which is is to be added. Also, since only one event is given, the keyword END is not needed. This form is intended to be used to easily add a single event to a table. An example of this form would be:

```
ADD (LEFT,SOFA,21,14.2,0) TO T
```

The third form of the ADD instruction is identical to that of the second except that instead of giving the values of an event, the name of a previously defined event (defined using a DEFINE EVENT instruction) is given. The order of the descriptors (and their corresponding values) which was used in the DEFINE EVENT instruction need not be the same as that which was used in the DEFINE RT instruction. Not all descriptors in the relational table need appear in the event to be added (the corresponding values are left undefined); however, the event must not have any descriptors which are not in the relational table. Thus, to add the event E1 (as defined above) the user would enter the instruction:

ADD E1 TO T

NOTE: in this case, the value of the descriptor E for this event is undefined.

Normally, the event or events to be added by any of the three forms of the ADD instruction are inserted at the end of the table. The user may, however, indicate where in the table the event is to be added by specifying before or after which row (indicated by the row number) he wishes it to be added. This specification is in the form of a simple VL1 condition (only one selector is necessary). For example, to add the event E1 to the beginning of table T, the instruction would be:

```
ADD E1 TO T:[ROW < 1]
```

There are two retrieval instructions, GET and LET. They each create a new relational table from data in one or more relational tables. The relational table created in a GET instruction is printed out at the user's terminal, while the table created by a LET instruction is only stored. A label must be assigned to the table created by a LET instruction; this label becomes the name of the relational table. In a GET instruction, the label is optional; if it is given, then the retrieved table is retained (by the system) under the name given as the label; if it is not given, then the table is destroyed after being printed out. (This is the reason the label is required for the LET, since a label-less LET would not serve any purpose; the LET is used to create a table, to be used later, without having it printed out.) Those tables created by GET instructions which are labeled and all those created by LET instructions are treated as those relational tables created by DEFINE and ADD instructions; the exception to this is that those created by retrieval disappear at the end of the session unless the user indicates he wishes a particular table to be saved. The SAVE instruction is used for this purpose; the user types SAVE followed by the name of the (retrieved) table he wishes to save. Thus if NEWTAB had been a label on a GET or LET instruction, it would be made a permanent table in the

data base by the instruction:

```
SAVE NEWTAB
```

Following the (possibly optional) label, there is what shall be called a relational table condition, which is composed of two parts: a relational table expression and a condition on that expression. The table expression specifies which descriptor or descriptor from which table or tables are to be used in the retrieval. If the descriptors are from more than one relational table, then the descriptors are extracted from the join of the tables specified in the retrieval. The join of two relational tables (which must share a common descriptor) is defined as a new relational table having as descriptors the union of the sets of descriptors from the two tables and each of whose events is formed by concatenating an event from the first table with an event from the second table which has the same value for the common descriptor (if for a given value of the common descriptor the first table has M events with that value for the common descriptor and the second table has N events with that value for the common descriptor, then the joined table has M*N events with that value for the common descriptor). The join of K tables (K > 2) is defined as the join of the first table with the result of joining the last K-1 tables. Specifying a subset of the

descriptors of a relational table (the join of two or more relational tables) causes the formation of the projection of the full table (joined table) by extracting only those descriptors specified. The user may also form an expression using the set theoretic operators "+" (union), "*" (intersection), and "-" (difference); in these cases the two tables being operated on must have the same descriptors and in the same order.

If no descriptors are specified (i.e. the name of a relational table, or the union, interection, or difference of two relational tables, is given without specifying any of its descriptors), then the full set of descriptors from that table is retrieved. Alternatively, instead of specifying some descriptor, a function of that descriptor may be specified. The available functions are: MIN (minimum), MAX (maximum), AVG (average), SUM, COUNT (cardinality), and DOMAIN (the set of all values--from any relational table--which are in the domain of that descriptor). Rather than retrieving all the values for that descriptor, the specified function is applied to that descriptor, and the result of that function is what is "retrieved". The following are all valid relational table expressions:

T	All of table T
T(A,B)	Descriptors A and B from table T
(T.C)	Descriptor C from table T

(T.C,TT.X)	Descriptors C and X from the join of tables T AND TT
T+U(A,C)	Descriptors A and C from the union of tables T and U
T*U(B)	Descriptor B from the intersection of tables T and U
(SUM(T.A),AVG(T.B))	

The condition, which is optional, specifies which events from the relational table expression are to be retrieved (all of them, if no condition is specified). The condition is separated from the relational table expression by a colon, which has the meaning "such that". The form of the condition is taken from that of a formula in the VL1 logic system[3]. A condition is the disjunction of one or more terms, each of which is the conjunction of one or more selectors. In VL1 a selector has three parts: the referee, which is the name of a descriptor, a comparison operator ("<", "<=", "=", "NOT =", ">=", or ">"), and the reference, which is a list, each of whose elements is either a single value or a range (which is denoted by the lower bound, a colon, and an upper bound). A selector is said to be "satisfied" if the value of the descriptor in the referee has the proper relationship (indicated by the comparison operator) to the values in the reference (for "=", it is satisfied if the value of the descriptor is is equal to a

value--or within some range of some range--in the reference; for "NOT =", it is satisfied if the value of the descriptor is not equal to any of the values--and is not within any range of values--in the reference; for the other operators, the reference usually only has one value, in which case the value of the reference is compared to that value). Examples of selectors are:

```
[A=5]
[B>7]
[C NOT= TOP,BOTTOM]
[T.E<=7.3]
```

In the condition of the data base sublanguage, the form of the selector is somewhat expanded from that used in VL1. The form used in VL1 corresponds to the first form of the selector in the sublanguage; however, the sublanguage has two additonal forms. The first of these allows a relational table condition as the referee and another as the reference. In this case the comparison operators become set theoretic comparison operators, where the relational table conditions are treated as sets of events (which they really are), and with "<", "<=", ">=", and ">" denoting subset and superset operations. Examples are:

```
[ (T.A)=(NEWT.A) ]
```

```
[(B) <= TT+U(F):[F>10]]
```

The first example tests whether or not the relational table composed of column A from the table T has the same values as column A from the relational table NEWT (i.e. the two columns, treated as sets, are compared for set equality). The second example tests whether or not the relational table composed of column B from relational table T (which need not be specified if it is the only table with B as a descriptor) is a subset of the relational table which is composed of all values of the descriptor F from tables TT and U (i.e. from the union of TT and U) which are greater than 10.

In the final form of the selector, the referee is a single event and the reference is a relational table condition. The comparison operators allowed are IN and NOT IN, and the comparison is whether or not the event specified as the referee is in (not in) the relational table specified by the relational table expression in the reference. Some examples of selectors of this form are:

```
[(3,2,5.2] IN U]
[(TABLE,ARM) NOT IN T(C,D):[T.A=5]]
```

A term is satisfied if and only if each of its selectors is satisfied, and a condition is satisfied if and only if at least one of its terms is satisfied. For each

event in the relational table expression, the condition is applied to the values of that event; if the condition is satisfied, then that event is retrieved and is included in the resultant relational table.

There is another type of relational table condition which is called the image set. It is the set of all values of a descriptor (this set is really a relational table) which appear in events with other given values of descriptors in the same relational table. Thus

(A:B=5)

is the set of all A's such that B is 5. Another useful form is where the descriptor (or descriptors) to the right of the colon is not given specific values but is a descriptor being retrieved, for example:

(A:B)

where B is a descriptor being retrieved, is the set of all A's corresponding to that B.

As can be seen, the retrieval instructions have quite sophisticated capabilities. As a result they may sound complicated; however, most retrievals are simple and can be expressed simply. Usually only the descriptors from one

table are specified; usually the condition has only one term and this term has only one or two selectors; usually the selectors are of the first form and only use the comparison operators "=" or "NOT =" with only one value in the reference. Thus a typical statement would be:

```
GET T(A,B):[B=5]
```

More complicated retrievals can frequently be simplified by breaking them into several LET instructions followed by a GET (this does not violate the principle of non-procedurality, which will be discussed in Chapter 3; the user is just specifying the order in which the retrievals are to be done, and this order may more closely correspond to the order in which he thinks of the query); more sophisticated users will like the flexibility of the condition, and with a little experience will be able to specify any retrieval he wishes in a single GET. The user has the freedom to choose which of these methods he prefers. A more complicated instruction would be:

```
GET TB:=T-U(A):[B=2,10][D NOT=LEG] OR [B<5] OR
                [C=YES,FIVE]
```

A GET instruction is usually used to print out part (or all) of a single relational table; the table T would be printed using the instruction:

```
GET T
```

A LET is usually used to make a copy of part (or all) of an existing relational table or to store a temporary result to be used in the next retrieval instruction; a copy of the table T could be made using the instruction:

```
LET NEWT:=T
```

Both the GET and LET, however may be used to perform more complicated retrievals. More examples are provided in the next chapter.

There are two instruction which are used to alter the existing data in the data base. One is the CHANGE instruction, which is used to modify given entries in relational tables by replacing current values by new values; the structure of the data base is not changed at all. The other modify instruction is the DELETE, which deletes current events or descriptors (or any combination of events and descriptors) from an existing relational table.

The CHANGE instruction is really a compound instruction. It is delimited by a CHANGE statement and and END statement. The CHANGE statement specifies what subtable, or list of subtables, may be modified within the CHANGE instruction. Each subtable is specified by a relational table condition. Although this expression may be as general as that used in a retrieval instruction, only those expressions which extract a subtable from one existing relational table make sense. Specifying a subtable to be changed which is derived from more than one relational table is analogous to passing an expression (in some programming language) as a parameter to a procedure and having the procedure modify the formal parameter for the purpose of modifying the actual parameter: there is no corresponding variable in the calling routine to modify! The CHANGE statement in the CHANGE instruction causes the creation (retrieval) of a subtable or list of subtables. These subtables are operated on (changed), within the CHANGE instruction, in the user's workspace; at the end of the CHANGE instruction, the modified values are copied back into the relational table from which it was extracted (if the subtable had been extracted from the join of two or more tables, there would be no corresponding table to return the changed values to; the join is just a calculated table which is derived from values in the data base but does not really exist as part of the real data base). Within the

CHANGE instruction the user may use assignment statements to modify existing entries in that subtable (or subtables). Each assignment statement may optionally have a condition modifying it, specifying for which events that descriptor (which is being assigned to) is to be modified (i.e. an entire column may be modified). If the condition specifies row 0, then the name of the descriptor is modified in that table. Within the CHANGE instruction, in addition to the assignment statement, the user is allowed to use a DISPLAY statement and a GET statement, which is restricted form of the GET instruction. The DISPLAY statement is used to display the current value of the subtable (or subtables) being changed. The GET statement is restricted by not allowing it to have a label; its purpose is only for that of display (e.g. to display the values in the original table from which the changed subtable was extracted) and not for the purpose of creating new tables. To terminate the CHANGE instruction, the user enters either the keyword END or the keyword ABORT. END causes the subtable (or subtables) in the user's workspace to be copied back into the original table (or tables); ABORT prevents this updating from being done. A sample instruction would be:

```
CHANGE T(A,B):[B>2.3]   change only descriptors A and B,
                        and only for those rows where
                        B is greater than 2.3
    B:=B+1.2            add 1.2 to B in every row
    A:=5 : [A<9]        change the value of A to 5 in only
```

```
                                  those rows where A is less than 9
    DISPLAY                       now display the table
    GET T(A,B):[B>2.3]            display the original subtable
    A:=3 : [ROW=10]               change the value of descriptor A
                                  in row 10 to the new value 3
END
```

The DELETE instruction is used to extract and dispose of a subtable from an existing relational table (the subtable may be the entire table). The subtable to be deleted is specified by a relational table condition, but as in the CHANGE instruction, the specification of only the subtable from one existing relational table makes any sense. If an entire column is specified (i.e. a descriptor given without a condition), then that entire column is removed from the table, and the table now has one descriptor fewer. If only some of the events for a given descriptor are specified, then those entries in the table are left undefined. If all the descriptors for a given event are deleted, then the event is removed from the table. If a relational table is specified without specifying any descriptors or a condition on the rows, then the entire relational table is disposed of (and its name is freed and may be reused for any purpose). Thus the DELETE instruction can serve any of four different purposes:

1. To destroy an entire relational table, for example:

```
DELETE T
```

2. To delete a set of descriptors from a relational table, for example:

   DELETE T(A,C)

3. To delete a set of events from a relational table, for example:

   DELETE T:[ROW=2,10] OR [A=5]

4. To erase certain entries in a relational table, for example:

   DELETE T(B,D): [B>7][D=ARM]

All four are specified using the same basic form but differ in their specification of what to delete.

The INDUCE instruction is used to induce a set of VL decision rules for a descriptor based on knowledge which the system has stored in tables. This knowledge may be in the form of relational tables of event sets, tables of previously derived rules (either induced or fed into the system), or any combination of event sets and rules. For each relational table, the user may specify which descriptor is to be used as the class specification (the value of that descriptor in each event is the class to which the event belongs) or that all events from that table belong to a specified class (which does not correspond to any descriptor in the table). If no classes are given for any of the tables, and if the descriptor which is to be induced is not in any of the relational tables, then the rule which is induced covers the set of events specified against its

inverse (this is called UNICLASS induction [5]). If there is a mixture of event sets and rules, then the rule formed is by feedback learning [6]. The user may specify what induction technique to use in forming the rule; if none is given, and none of the above cases applies, then the standard AQVAL method [1,7] is used by default. The options which may be specified are SYMMETRIC [8] and AQVAL; the user may also specify either of UNICLASS or FEEDBACK, but these should not be necessary since the system can figure out which of these to use based on which types of tables are specified. The user may also specify values of parameters to be used in running the induction program; the values of all other parameters are either by default or entered by the user in a program-driven conversational mode (i.e. the program asks for each value individually). Some examples of this instruction are:

```
INDUCE R1:=C USING T
INDUCE SYMMETRIC R2:=DISEASE USING T(CLASS=1),U(CLASS=2)
INDUCE R4:=A USING R3(RULE),T(CLASS=A)
```

The first instruction forms a rule for describing descriptor C of table T using the default (AQVAL) method and labels the new rule table "R1". The second instruction forms a symmetric rule for a new descriptor called DISEASE using the table T for examples of DISEASE=1 and the table U for

examples of DISEASE=2; the resultant rule table is called "R2". The third example used the feedback learning method to form a new rule table called "R4" updating the old rule table R3 by using new facts from the table T, which uses descriptor A to specify the class of each event.

The DEDUCE instruction asks that the value of an unknown descriptor be deduced given a table of rules and optionally the values of known descriptors. These values may either be given explicitly or else may come from a defined event (defined using the DEFINE EVENT instruction). If the value of the descriptor cannot be deduced with the given information, the system indicates this and may ask for the required information, if it is known to the user. An example of the DEDUCE instruction is:

```
DEDUCE C FROM R USING (2,10,STOVE)
```

Here a previously derived rule R is used to deduce the value of C using the event "(2,10,STOVE)"; the order of these values corresponds to the order of the descriptors used originally in forming R.

There are two more instructions which do not make use of the data base but are sometimes quite useful; these are COMMENT and HELP. COMMENT allows the user to enter a comment in order to document (for his own purposes) what he

is trying to do. The program ignores all text up to the symbol END. A comment may also be entered (PL/1 style) between any 2 symbols by enclosing it within the delimiters "/*" and "*/".

HELP asks for help; the user may either ask for an English language explanation of an instruction by giving the instruction name or may ask for the production rule from the grammar for the sublanguage for any nonterminal in the grammar (see Appendix A for a listing of the grammar). For example:

```
HELP GET
```

provides a description of the GET instruction, while

```
HELP <CONDITION>
```

provides the production rule associated with the nonterminal <CONDITION> (which is the nonterminal which derives all VL conditions).

3. COMPARISON WITH OTHER SUBLANGUAGES AND EXAMPLES

The VL Relational Data Sublanguage is modelled after the relational sublanguage ALPHA [4] and the Variable-Valued Logic language VL1 [3]. VL1 was chosen as a model for two reasons. The first is that the sublanguage is intended to be a subset of a full language for communication with the Inferential Computer Consultant. The basis for the consultant is variable-valued logic and its language VL1. Since VL1 must be used in other aspects of the system--rule formation (induction) and rule processing (deduction)--a VL1 based data sublanguage is the natural choice for the sake of consistency. In addition to this, VL1 very conveniently expresses the conditions which must be specified in data base operations (other sublanguages frequently call these conditions "predicates", since they are taken from Boolean logic). Since VL1 is the variable-valued logic extension of Boolean logic, much of the VL sublanguage is already similar to the data sublanguages which are based on Boolean logic, which include ALPHA. This similarity is most notable in simple retrievals, where the full power of the VL sublanguage is not required; however, for more complicated retrievals, the VL sublanguage simplifies the specification of what the user wishes to retrieve. There are several reasons that the VL sublanguage is often easier to use. One is that a VL1 selector allows a list of values in the

reference. Most other sublanguages only allow predicates in which the value of a descriptor (which ALPHA calls a domain) may be compared with a single value; in these other languages, if all rows having any of several values for a given descriptor are desired, then a disjunction of predicates (if the sublanguage even allows disjunction) must be used, whereas in the VL sublanguage, a single selector may be used. This more closely corresponds to the English language, where one would say "x equals 1 or 3" rather than "x equals 1 or x equals 3". This simplification is a consequence of the use of variable-valued logic and the concept of a descriptor as a multi-valued logical variable.

A second simplification arises from the use of relational table conditions, which allows a secondary retrieval to be a subexpression of the primary retrieval (i.e. in the second and third forms of the selector, which allow a relational table condition to be in the referee--only in the second form--and the reference). This makes it easier to specify complicated retrievals in a single instruction. The sublanguages SEQUEL [9] and SQUARE [10] allow this sort of thing since they incorporate the idea of a relational expression, but the sublanguages QUEL [11] and ALPHA do not.

The third and most important simplification lies not in the actual syntax of the sublanguage but in the specification of the use of the sublanguage. In all the other languages, the links between tables must be explicitly specified. That is, if tables T1 and T2 both have a common descriptor A and the user wishes to refer to a descriptor B1 in T1 corresponding (through common values of descriptor A) to a descriptor B2 in T2, then the query will usually be required to use the predicate:

T1.A=T2.A

The VL sublanguage does not require this (although it may be specified if the user wishes to aid the system in finding this link); instead these links are automatically determined by the system (in fact, links longer than one table are also allowed to be implicit). This simplifies many retrievals and also simplifies the sublanguage: ALPHA and QUEL had to introduce the RANGE statement and ALPHA also existential quantification for this purpose only.

The remainder of this chapter will compare, through examples, the VL data sublanguage with the sublanguages ALPHA, SEQUEL, SQUARE, and QUEL. All these languages are based the relational calculus rather than relational algebra. That is, one does not specify, in a query, how to perform the retrieval (i.e. what operations must be

performed) but what are the characteristics of the data which is to be retrieved (and the system takes care of determining what operations must be performed). This approach is easier for non-mathematically oriented users of the sublanguage (and frequently for mathematically oriented users). The specification of what to retrieve more closely corresponds to the thought process which a user must go through when he decides what he wants retrieved. Thus these "non-procedural" sublanguages will be the only ones considered here.

The relational tables used in the following examples are taken from Figure 4.1 on page 64 of Date[12]; some of the examples are also taken from Chapter 4 of Date. These tables could be constructed in the VL sublanguage using the following instructions:

```
COMMENT DEFINE THE DESCRIPTORS AND KEY OF THE SUPPLIER RT
END

DEFINE S(S#,SNAME,STATUS,CITY)KEY:=S#
END

ADD TO S
(S1,SMITH,20,LONDON)
(S2,JONES,10,PARIS)
(S3,BLAKE,30,PARIS)
(S4,CLARK,20,LONDON)
(S5,ADAMS,30,ATHENS)
END

COMMENT DEFINE THE RT-S P AND SP END

DEFINE P(P#,PNAME,COLOR,WEIGHT) KEY:=P#
       SP(S#,P#,QTY) KEY:=S#,P#
END

COMMENT NOW FILL IN THE TABLE P   END
```

```
ADD TO P
 (P1,NUT,RED,12)
 (P2,BOLT,GREEN,17)
 (P3,SCREW,BLUE,17)
 (P4,SCREW,RED,14)
 (P5,CAM,BLUE,12)
 (P6,COG,RED,19)
END

COMMENT ADD EVENTS TO SP END

ADD TO SP
 (S1,P1,3)
 (S1,P2,2)
 (S1,P3,4)
 (S1,P4,2)
 (S1,P5,1)
 (S1,P6,1)
 (S2,P1,3)
 (S2,P2,4)
 (S3,P3,4)
END

ADD TO SP
 (S3,P5,2)
 (S4,P2,2)
 (S4,P4,3)
 (S4,P5,4)
END

ADD (S5,P5,5) TO SP;
```

These tables may then be printed out using the following GET instructions (the following is taken from an actual run of the program implementing the VL language):

```
GET S;
```

S

S#	SNAME	STATUS	CITY
S1	SMITH	20	LONDON
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S4	CLARK	20	LONDON
S5	ADAMS	30	ATHENS

```
GET P;
```

P

P#	PNAME	COLOR	WEIGHT
P1	NUT	RED	12
P2	BOLT	GREEN	17
P3	SCREW	BLUE	17
P4	SCREW	RED	14
P5	CAM	BLUE	12
P6	COG	RED	19

```
GET SP;
```

SP

S#	P#	QTY
S1	P1	3
S1	P2	2
S1	P3	4
S1	P4	2
S1	P5	1
S1	P6	1
S2	P1	3
S2	P2	4
S3	P3	4
S3	P5	2
S4	P2	2
S4	P4	3
S4	P5	4
S5	P5	5

Once these tables are created (whatever facilities the other sublanguages have for creating relational tables), they can be manipulated as follows in these examples (the formulation of the queries in SEQUEL, SQUARE, and QUEL, and some of those of ALPHA are those of the author, who apologizes for any misrepresentation of these sublanguages):

1. Goal: Retrieve from S the supplier number and status of suppliers in London. Label the result W.

```
VL        GET W:=S(S#,STATUS):[CITY=LONDON]

ALPHA     GET W(S.S#,STATUS):CITY='LONDON'

SEQUEL    W:SELECT S#,STATUS
            FROM   S
            WHERE  CITY='LONDON'

SQUARE      W<-              S      ('LONDON')
                 S#,STATUS   CITY

QUEL      RANGE: S(X)
          RETRIEVE: W:X.S#,X.STATUS:X.CITY='LONDON'
```

2. Goal: Retrieve all of S

```
VL        GET S
           or
          GET S(S#,SNAME,STATUS,CITY)

ALPHA     GET W(S)
            or
          GET W(S.S#,S.SNAME,S.STATUS,S.CITY)

SEQUEL    SELECT S

SQUARE                              S
          S#,SNAME,STATUS,CITY

QUEL      RANGE: S(X)
          RETRIEVE: W:X.S#,X.SNAME,X.STATUS,X.CITY:
```

3. Goal: Retrieve supplier number of all suppliers in Paris whose status is greater than 20

```
VL        GET (S.S#):[CITY=PARIS][STATUS>20]

ALPHA     GET W(S.S#):S.CITY='PARIS' ∧ S.STATUS>20

SEQUEL    SELECT S#
          FROM   S
          WHERE  CITY='PARIS'
          AND    STATUS>20

SQUARE      S            ('PARIS',>20)
          S# CITY,STATUS

QUEL      RANGE: S(X)
          RETRIEVE: W:X.S#: (CITY='PARIS') ∧ (STATUS>20)
```

4. Goal: retrieve supplier number and status of suppliers in Paris, in descending order of STATUS

```
VL        GET S(S#,STATUS):[CITY=PARIS]
               ORDER DOWN ON STATUS

ALPHA     GET W(S.S#,S.STATUS):S.CITY='PARIS'
                 DOWN S.STATUS

SEQUEL    impossible

SQUARE    impossible

QUEL      impossible
```

5. Goal: Retrieve supplier numbers of suppliers who supply part P2

```
VL        GET SP(S#):[SP.P#=P2]

ALPHA     GET W(SP.S#):SP.P#='P2'
```

```
SEQUEL    SELECT S#
          FROM   SP
          WHERE P#='P2'

SQUARE      SP   ('P2')
          S#  P#

QUEL      RANGE: S(X)
          RETRIEVE: W:X.S#:X.P#='P2'
```

6. Goal: Retrieve names of suppliers who supply part P2

```
VL        GET (SNAME):[P#=P2]
             or
          GET (S.SNAME):[SP.P#=P2]

ALPHA     RANGE SP X
          GET W(S.SNAME):∃X(X.S#=S.S# ∧ X.P#='P2')
SEQUEL    SELECT SNAME
          FROM   S
          WHERE  S# = SELECT S#
                      FROM   SP
                      WHERE  P#='P2'

SQUARE          S     ∘    SP   ('P2')
          SNAME S#     S#  P#

QUEL      RANGE: S(X):SP(Y)
          RETRIEVE: W:X.SNAME:(X.S#=Y.S#) ∧ (Y.P#='P2')
```

7. Goal: Retrieve names of suppliers who supply red parts

```
VL        GET(SNAME):[COLOR=RED]

ALPHA     RANGE P X
          RANGE SP Y
          GET W(S.SNAME):
            ∃X ∃Y(S.S#=Y.S# ∧ Y.P#=X.P# ∧ X.COLOR='RED')
```

```
SEQUEL    SELECT SNAME
          FROM   S
          WHERE S# = SELECT S#
                     FROM   SP
                     WHERE  P# = SELECT P#
                                 FROM   P
                                 WHERE  COLOR='RED'


SQUARE          S     •    SP    ◦    P       ('RED')
          SNAME S#        S#  P#       P# COLOR


QUEL      RANGE: P(X):SP(Y):S(Z)
          RETRIEVE: W:z.sname:
            (Z.S#=Y.S#) ∧ (Y.P#=X.P#) ∧ (X.COLOR='RED')
```

8. Goal: Retrieve the names of the suppliers who supply at least one part supplied by S2

```
VL        GET (SNAME):[COUNT((P#:SNAME)*(P#.S#=S2)>0]
          COMMENT (P#:SNAME) is the set of all P# associated
             with the SNAME being considered for the
             retrieval.  (P#:S#=S2) is the set of all P# for
             supplier S2. '*' denotes intersection of
             these two sets (relational tables)
          END
```

or

```
          GET (SNAME):[ (P#) <=(P#:S#=S2) ]
          COMMENT '<=' checks for set inclusion (subset) END


ALPHA     RANGE SP X
          RANGE SP Y
          GET W(S.SNAME): ∃X(X.S#=S.S#)
                              ∧ ∃Y(Y.P#=X.P# ∧ Y.S#='S2'))


SEQUEL    SELECT SNAME
          FROM   S
          WHERE  S# = SELECT S#
                      FROM   SP
                      WHERE  P# = SELECT P#
                                  FROM   SP
                                  WHERE  S#='S2'
```

```
SQUARE          S    •    SP    •    SP   ('S2')
           SNAME S#    S#   P#    P#   S#
```

```
QUEL       RANGE: S(X):SP(Y,Z)
           RETRIEVE: W:X.SNAME: (X.S#=Y.S#)
                                ∧ (Y.P#=Z.P#) ∧ (Z.S#='S2')
```

9. Goal: Retrieve all part numbers and their corresponding cities

```
VL         GET (SP.P#,S.CITY)
```

```
ALPHA      GET W(SP.P#,S.CITY):SP.S#=S.S#)
```

```
SEQUEL     SELECT P#,CITY
           WHERE  SP.P#=S.S#
```

```
SQUARE     TEMP           <-X   ⊂ S,SP
               P#,S#,CITY    S#
           TEMP
               p#,city
```

```
QUEL       RANGE: S(X):SP(Y)
           RETRIEVE: W:X.P#,X.CITY:X.S#=Y.S#
```

10. Goal: Retrieve names of all suppliers who supply all parts

```
VL         GET (SNAME):[ (P#:SNAME)=DOMAIN(P#) ]
```

```
ALPHA      RANGE P X
           RANGE SP Y
           GET W(S.SNAME): ∀X ∃Y (Y.S#=S.S# ∧ Y.P#=X.P#)
```

```
SEQUEL    SELECT SNAME
          FROM   S
          WHERE  S# = SELECT S#
                      FROM   SP
                      WHERE  P# ALL = ALL
                                      SELECT P#
                                      FROM  SP
```

```
SQUARE    x      ∈ S:
           SNAME
            SP (S   ) =   SUPPLY
          P#  S# S#     P#
```

```
QUEL      impossible
```

11. Goal: Retrieve the supplier numbers of those suppliers who supply at least all those parts supplied by S2

```
VL        GET (SP.S#):[ (P#:S#) >= (P#:S#=S2) ]
```

```
ALPHA     RANGE P X
          RANGE SP Y
          RANGE SP Z
          GET W(SP.S#): ∀X( ∃Y(Y.S#='S2' ∧ Y.P#=X.P#)
                          -> ∃Z(Z.S#=S.S# ∧ Y.P#=X.P#))
```

```
SEQUEL    SELECT S#
          FROM   SP
          WHERE  P# ALL >= ALL
                           SELECT P#
                           FROM   SP
                           WHERE S#='S2'
```

```
SQUARE    impossible
```

```
QUEL      impossible
```

The remaining examples demonstrate the VL CHANGE and DELETE instructions. SEQUEL and QUEL do not have such instructions and will not be included in the following:

12. Goal: Change the color of part P2 to yellow

```
VL          CHANGE (COLOR):[P#=P2]
               COLOR:=YELLOW
            END

                    or

            CHANGE P(P#,COLOR)
               COLOR:=YELLOW:[P#=P2]
            END

ALPHA       HOLD W(P.P#,P.COLOR):P.P#=P2
               W.COLOR=YELLOW
            UPDATE W

SQUARE      -> P          ('P2','YELLOW')
                P#;COLOR
```

13. Goal: Increase by 1 the quantity of each part supplied by S1

```
VL          CHANGE (QTY):[S#=S1]
               QTY:=QTY+1
            END

ALPHA       HOLD W(QTY):SP.S#='S1'
               W.QTY=W.QTY+1
            UPDATE W

SQUARE      -> SP         ('S1',1)
                 S#;QTY+
```

14. Goal: Multiply by 2 the quantity of each part supplied by S1 and make sure this number does not exceed 5; if it does, set it equal to 5

```
VL        CHANGE (QTY):[SP.S#=S1]
             QTY:=QTY*2
             QTY:=5 : [QTY>5]
          END

ALPHA     HOLD W(SP.QTY):SP.S#='S1'
             W.QTY=W.QTY*2
          UPDATE W
          HOLD W(SP.QTY) : SP.S#='S1' ∧ QTY>5
             W.QTY=5
          END

SQUARE    -> SP        ('S1',2)
               S#;QTY*
          -> SP             ('S1',>5,5)
               S#,QTY;QTY
```

15. Goal: Delete from S all suppliers in London

```
VL        DELETE S:[CITY=LONDON]

ALPHA     HOLD W(S):CITY='LONDON'
          UPDATE W

SQUARE    ↑ S     ('LONDON')
             CITY
```

16. Goal: Destroy the entire relational table P

```
VL        DELETE P

ALPHA     HOLD W(P)
          DELETE W

SQUARE    ↑ P
```

17. Goal: Remove the descriptor COLOR from P

VL DELETE (P.COLOR)

ALPHA impossible

SQUARE impossible

4. IMPLEMENTATION OF THE SUBLANGUAGE

The program implementing the VL data sublanguage is written in PASCAL using the PASREL compiler for the PDP-10 computer at the University of Illinois. The program is a set of externally compiled procedures which are called from the control program (also written in PASCAL) of the computer consultant. There are two major phases to the program: a parser, which is implemented by an external procedure called PARSE, and an execution package, implemented by an external procedure called EXEC. The parser converts each instruction entered by the user into an internal form which is passed to the execution package to actually execute the user's instruction. The separation into two phases serves two purposes. First, it follows the philosophy of structured and modular programming by keeping different functions in separate procedures (the functions of syntactic analysis, which is performed by PARSE, and the true processing of the instruction, which is performed by EXEC) and keeps the user interface localized. The second purpose is concerned with errors which may occur in the user's input; only syntactically correct instructions are passed to the execution package. This is quite important in the interactive environment of the computer consultant (as opposed to the environment of a batch or timesharing computer program which may be rerun after corrections are

made). If each instruction were executed up to the point of a syntax error--or if the execution supervisor attempted to ignore these errors or tried to execute what it thought the user meant--the data base might end up unalterably scrambled. For example, if the execution supervisor were in the middle of adding or deleting a relational table or an event but did not finish because of errors in the user's input, the internal data structures used to represent the data base might end up so jumbled that parts or all of the data base would be lost. The system cannot ensure that the user is always specifying exactly that which he intends, but it can attempt to protect the user as much as possible from errors. By parsing an entire instruction before beginning any execution of it, the system attempts to do this. A desirable side effect of the two phase process is that it would be possible for another module within the computer consultant to access the data base through the sublanguage by generating the appropriate internal text and passing it to EXEC. No such attempts have been made, but the possibility is being considered.

The parser is hard coded (as opposed to being table driven) and uses the recursive descent parsing technique. These choices were made in the interest of ease of implementation and debugging as well as ease of understanding by anyone reading the program (table driven programs and hard coded ones using other parsing techniques

are usually more difficult to follow). For each instruction (DEFINE, SAVE, ADD, GET, LET, CHANGE, DELETE, INDUCE, DEDUCE, and HELP), the source text is read in and the relevant information is encoded into a format suitable for processing by the execution supervisor. The data into which it is encoded is referred to as the internal text and is in the form of an integer array called PT (for parse table). The first three words of this array serve the same function for all instructions: the first word contains the length of the array as it was declared; the second contains the code for the instruction type; the third contains the length of the actual portion of the array used by the instruction. After this, each instruction has its own coding, although consistency is maintained whenever common structures are encountered. For example, the condition part of the GET, LET, CHANGE, and DELETE instructions are translated identically. All expressions (arithmetic, relational table, and VL) translated into a postfix string (contiguous elements of PT). Real numbers and strings are stored as indices into auxiliary arrays of the appropriate types (a real array called REALN and an array of strings--of length 20--called SYMTABLE). The complete meaning of the entries in PT is given in Appendix B.

During a session, the entire data base is stored in core. Although this places a restriction on the size of the data base, it was felt that the size and number of

relational tables one would want to use for the computer consultant would be within the limits that can be provided. The advantage to storing everything in core is that the entire data base can be quickly and easily accessed. Dynamic storage is used whenever possible so that only as much storage must be allocated as is needed. As a result of this approach there are few built-in limits.

Each relational table has a header node, and these nodes are storage as a linked list. The header node is implemented as a PASCAL record containing the following fields:

RTNAME--contains the name of the relational table. It is a pointer to a programmer defined data type called STRING, which is a record variant: PASCAL allows a field of a record to be one of several variants. When the record is allocated, the programmer specifies which one of these variants he wishes the field to be. By choosing the variants to be strings of various lengths (a string in PASCAL is an array of single characters, and the length of the array is the length of the string); thus only exactly the number of characters which are needed is allocated.

NEXTRT--a pointer to the header node for the next table in the linked list.

ROW0PTR--a pointer to the 0-th row (the row of descriptors) in the relational table.

ROWLASTPTR--a pointer to the last row of the relational table. This allows rows to be added to the end easily.

RTNUM--the relational table number, which is assigned by the program and is used for internal references to the table (rather than having to use the name).

NUMKEYS--the number of descriptors in the key of the table; the descriptors in the key are the first NUMKEYS descriptors in the table.

STATUS--indicates what type of table it is (used for the purpose of saving or deleting it). The following values mean the table has the given status:

-1:temporary (created by a LET or a labeled GET). The table will remain only until the end of the session, unless a SAVE instruction is used.

0:temporary (created by an unlabeled GET or as a result of a subexpression). It will be destroyed as soon as the program is through with it.

1:permanent table (created by a DEFINE or saved by a SAVE). It will be saved on disk when the session is over and read back in for the next session.

2:event (created by a DEFINE EVENT instruction and saved only until the end of the session).

A row in a relational table is also a record; the first field is NEXTROW, which is a pointer to the next row in the same table. The next field is a record variant, whose variants are integer arrays of various lengths. The length allocated is the number of descriptors. The tag field, which indicates which variant was chosen, contains the length of that array. This is why the header node does not need to store the number of descriptors in the table--each row contains that information.

The 0-th row contains the descriptor numbers for the descriptors in the table. There is a table of descriptors which is managed by another program within the computer consultant. This table associates with each descriptor a number which it uses for internal reference to that descriptor (in much the same manner as relational table numbers are used). The table of descriptors program can return a descriptor number given a descriptor name and visa versa. The NEXTROW pointer points to the first row of the table, which is the first event in the table. The integers in that row (and all subsequent rows) are value numbers, which are also associated with values (integers, reals, or names) by the table of descriptors program. The table of descriptors keeps a list of all the values within the domain of each descriptor and the program can associate value numbers with values and their corresponding descriptor names. Thus only integers are needed in the storage for a

relational table to represent the descriptors and their values.

The variables RTHEAD and RTTAIL point to the first and last header nodes, respectively, in the linked list of relational tables. These two variables are all that are needed (usually only RTHEAD) to access all of the relational tables; thus it is quite easy to pass the tables as parameters to external procedures (this is sometimes a problem in PASCAL since there are no external variables).

PARSE contains or uses the following procedures, whose descriptions are given below:

VLSCAN--obtains the next token from the user's input and classifies it. The parameter SYMBCLASS is given the value of the class of the symbol; the possible classes are RES (reserved word), RTN (relational table name), EVN (event name), DES (descriptor name), TXT (either a character string or an alphanumeric name which is neither of RES, RTN, EVN, or DES), INT (integer), RLN (real number), DBL (2 character delimiter, one of ":=", or "<=", ">="), DLM (1 character delimiter), or IGN (a comment, which is to be ignored). The token is returned in NXTSYMB as a string. For character strings and alphanumeric names, it is also returned as a longer string (20 characters) in the array LONGSYMB; if it is an integer, its value is returned in NXTINT; if it is a real, its value is returned in NXTREAL.

VLERR1--prints out an error message when a syntax error is found, indicating what token was erroneously input and what token was expected. It also sets NOERRORS to FALSE to inhibit execution of the instruction.

VLERR2--prints out a longer error message than VLERR1 and also sets NOERRORS to FALSE.

VERIFY--verifies that the current symbol matches the parameter SYMB, which is what is expected. If they do not match, then VLERR1 is called.

DESCINRT--determines if the descriptor number ND is in the relational table number NRT.

SCAN--calls VLSCAN with the appropriate external parameters. This is done so that each call to VLSCAN does not have to pass all the parameters (this would produce much more object code).

ADDSYMB--adds a string (the current symbol) to SYMTABLE.

LOOKRELOP--determines if the current symbol is a relational operator.

GETDESC--obtains the descriptor number and relational table number of the current (and following) symbol.

VALUE--parses a value, which may be an arithmetic expression or a name. It fills in PT with the postfix translation of the expression.

CONSTANT--(within VALUE) parses a single constant, which is an integer, a real number, or a name.

AFACTOR--(within VALUE) parses an arithmetic factor.

ATERM--(within VALUE) parses an arithmetic term.

RTCOND--parses a relational table condition, which consists of a relational table expression and (optionally) a VL condition.

CONDITION--parses a VL condition and fills in PT with the postfix translation of the value.

SELECTOR--(within CONDITION) parses a selector.

VLTERM--(within CONDITION) parses a VL term.

VLDEFINE--parses a DEFINE instruction

DEFRT-(within VLDEFINE) parses the definition of one relational table.

VLADD--parses an ADD instruction

ADDVAL--(within VLADD) parses a list of event in an ADD instruction.

ADDTO--(within VLADD) parses the relational table name to which the event or events are to be added and the row condition (if any), which specifies where in the table the event or events are to be added.

VLCHANGE--parses a CHANGE instruction.

CHANGE--(within VLCHANGE) parses a single CHANGE statement.

VLDELETE--parses a DELETE instruction.

VLHELP--parses a HELP instruction.

VLCOMMENT--parses a comment.

VLSAVE--parses a SAVE instruction.

STMTYPE--determines the type of instruction and sets the code in the variable NXTKEYWD.

PFULL--an external procedure from the table of descriptors program which fills in (in the table of descriptors) the name of a new descriptor.

TODNEW--an external procedure from the table of descriptors program which allocates storage for a new descriptor.

EXEC contains or uses the following procedures, whose descriptions are given below:

NEWSTR--a function which, given a string stored as an array of twenty characters returns a pointer to a record of type STRING with only the number of characters allocated as are needed (trailing blanks are trimmed off).

FINDRT--a function which, given a relational table number, returns a pointer to the header node for that table.

FINDROW--a function which, given a relational table number and the number of a row in that table, returns a pointer to the row.

PRTVAL--prints out a record of type TVALUE (which has as record variants the types INTEGER, REAL and STRING).

ALLOC--allocates a new row of length LENGTH for a relational table. This procedure is used rather than directly calling the built-in procedure NEW, which ALLOC calls, since NEW requires as parameter a constant to indicate which variant to allocate, while ALLOC allows this parameter to be a variable (or an expression).

SALLOC--allocates a STRING of length LENGTH; this procedure is needed for the same reason ALLOC is.

NEWRTN--allocates and fills in a header node for a new relational table.

CONDITION--evaluates a VL condition and determines whether or not it is satisfied for a given event.

MATCH--determines whether or not the key in a new event to be added to some relational table matches the key of an event already in the table.

DEFINE--executes a DEFINE instruction. For DEFINE RT, it allocates and fills in a new header node. Also, the 0-th row is allocated and filled in with the appropriate descriptor numbers. For a DEFINE EVENT instruction, an event is treated as a relational table with only one row. In addition to allocating and filling in the header node and the 0-th row, the first row, which is the event, is allocated and filled in.

ADD--executes an ADD instruction. It adds a new row (or rows) to a relational table (after insuring that there is no other row in that table with the same key value) with the appropriate value numbers. If a defined event is to be added, the the values for the event are first rearranged (if necessary) to match the order of the descriptors in the relational table.

DELETE--executes a DELETE instruction. If an entire relational table is to be deleted, then the header node is removed from the linked list of such nodes; this effectively frees the relational table name to be used for any purpose. If a set of descriptors are to be deleted, then the entire table must be recopied saving only those descriptors not deleted. The descriptors are not removed from the table of descriptors since they may be in use as descriptors of other relational tables. If a set of events is to be deleted, then they are just removed from the table. If the intersection of a set of descriptors or events are to be deleted (erased), then the appropriate entries are set to be undefined.

GETLET--executes a GET or a LET instruction.

GETSUBSET--(within GETLET) forms a new relational table with the subset of the original table (or join of two or more tables) which satisfies the condition and the subset of descriptors which were specified.

DISPLAY--(within GETLET) prints out the resultant table from a GET instruction).

GETMAP--(within GETLET) sets up a map of the descriptor numbers from the original table (or join of tables) to the retrieved table.

SAVE--executes a save instruction by setting the status of the relational table to "permanent" so that it is saved on disk at the end of the session.

PCARD--an external function from the table of descriptors program which returns a value number given a value and a pointer to the corresponding descriptor.

NFIND-an external function from the table of descriptors program which returns a pointer to a descriptor given the descriptor number.

VALFDP--an external procedure from the table of descriptors program which returns a descriptor value given a pointer to the descriptor and the value number.

The sequence of actions which the program performs when an instruction is entered by the user is as follows: First, PARSE is called to parse the instruction. Within PARSE, first STMTYPE is called to determine what type of instruction it is. This sets the instruction type variable, NXTKEYWD, and the appropriate parsing procedure is called. If there are no errors in the instruction (if PARSE returns with NOERRORS having the value TRUE), and if the instruction is one which requires execution (COMMENT and EXIT do not), then EXEC is called with the internal text. EXEC first

determines what type of instruction it was (from PT[2]) and calls the appropriate procedure to execute the instruction. The instruction is then executed.

REFERENCES

[1] Michalski, R.S., "Learning by Inductive Inference," NATO Study Institute on Computer-oriented Learning Processes, August 26-September 7, 1974, France (invited paper).

[2] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," CACM 13, No. 6, June 1970.

[3] Michalski, R.S., "VARIABLE-VALUED LOGIC: System VL1," 1974 International Symposium on Multiple-valued Logic, West Virginia University, Morgantown, West Virginia, May 29-31, 1974.

[4] Codd, E.F., "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

[5] Yuen, H., "UNICLASS Cover Synthesis: User's Guide," Internal document.

[6] Larson, James, "A Multi-Step Formation of Variable Valued Logic Hypotheses," Sixth Annual International Symposium on Multiple-Valued Logic at Utah State University, May 25-28, 1976.

[7] Michalski, R.S., and James Larson, "AQVAL/1 (AQ7) User's Guide and Program Description," Report No. 731, Department of Computer Science, University of Illinois, Urbana, Illinois, June, 1975.

[8] Jensen, Gerald M., "The Determination of Symmetric VL1 Formulas: Algorithms and Program SYM-4," M.S. Thesis, Department of Computer Science, University of Illinois, December, 1975.

[9] Chamberlin, Donald D., and Raymond F. Boyce, "SEQUEL: A Structured English Query Language," Proc. 1974 ACM SIGFIDET Workshop, Ann Arbor, Michigan.

[10] Boyce, R.F., Donald D. Chamberlin, W. Frank King III, and Michael M. Hammer, "Specifying Queries as Relational Expressions: SQUARE," IBM Technical Report RJ 1921, IBM Research Laboratory, San Jose, California, October, 1973.

[11] McDonald, Nancy, Michael Stonebraker, and Eugene Wong, "Preliminary Design of Ingres," Memorandum No. ERL-435, University of California, Berkeley, California, April 19, 1974.

[12] Date, C.J., An Introduction to Database Systems, Addison-Wesley Publishing Company, 1975.

APPENDIX A

GRAMMAR FOR VL RELATIONAL DATA SUBLANGUAGE

The following is a complete syntax specification for the VL sublanguage. Anything enclosed in [and] is optional; anything enclosed in { and ...} may be repeated 0 or more times.

```
SESSION        ::= {<VLI>; ...} EXIT

VLI            ::= <CREATE> | <RETRIEVE> | <MODIFY>
                 | SAVE <RTNAME> | COMMENT {any text} END
                 | <INDUCE> | <DEDUCE> | <HELP>

CREATE         ::= <DEFINE_INS> | <ADD_INS>

DEFINE_INS     ::= DEFINE <DEF_LIST> END

DEF_LIST       ::= {RT} <RTNAME> (<DESC> {,<DESC>...})
                     {KEY:=<DESC>{,<DESC>...}}
                     {<RTNAME>(<DESC>{,<DESC>...})
                     {KEY:=<DESC>{,<DESC>...}}...}
                 | EVENT <EVENTNAME>:=
                    (<DESC>:=<VALUE>{,<DESC>:=<VALUE>...})
                   {<EVENTNAME>:=

(<DESC>:=<VALUE>{,<DESC>:=<VALUE>...})...}

ADD_INS        ::= ADD TO <RTNAME>{:<ROW_COND>}
                       (<VALUE>{,<VALUE>...})
                      {(<VALUE>{,<VALUE>...})...}
                   END
                 | ADD <EVENTNAME> TO <RTNAME>{:<ROW_COND>}
                 | ADD (<VALUE>{,<VALUE>...}) TO <RTNAME>
                       {:<ROW_COND>}

ROW_COND       ::= [ ROW <GT> <ROW_VAL> ]
                 | [ ROW <LT> <ROW_VAL> ]

LT             ::= <

GT             ::= >

ROW_VAL        ::= <VALUE> | LAST

MODIFY         ::= CHANGE {RT} {<LABEL>:=} <RTCOND>
                              {,{<LABEL>:=} <RTCOND> ...}
                     <CHANGE_STMT>{<CHANGE_STMT>...}
```

```
                  <END_CHANGE>
                | DELETE {RT} <RTCOND> {,<RTCOND> ...}

END_CHANGE      ::= END | ABORT

LABEL           ::= <RTNAME>

CHANGE_STMT     ::= <DESC>:=<VALUE>{:<CONDITION>}
                  | <DESC>:=<NEWNAME>:[ROW=0]
                  | GET <RTCOND> {<ORDER>}
                  | DISPLAY {<RTNAME>}

RTCOND          ::= <RT> {<CSYM> <CONDITION>} | ISET

CSYM            ::= : | WHERE

CONDITION       ::= <VL_TERM> {OR <VL_TERM>...}

VL_TERM         ::= <SELECTOR> {<SELECTOR>...}

SELECTOR        ::= [<DESC> <RELOP> <VALUES>{,VALUES>...}]
                  | [<RT_COND> <RELOP> <RT_COND>]
                  | [<VALUE>{,<VALUE>...} {NOT} IN <RT_COND>]

RELOP           ::= <LT> | <LT>= | = | NOT= | <GT>= | <GT>

VALUES          ::= <VALUE> {:<VALUE>}

VALUE           ::= <AEXPR> | <ISET> | <NAME> | ?

  Note: "?" means the value is unknown; it is left undefined

ISET            ::= (<DESC> <CSYM> <DESC> {=<VALUE>}
                                  {,<DESC> {=<VALUE>}...})

AEXPR           ::= {<AEXPR> +} <ATERM> | <AEXPR> - <ATERM>

ATERM           ::= {<ATERM> *} <AFACTOR>
                  | <ATERM> / <AFACTOR>

AFACTOR         ::= <CONSTANT> | (<AEXPR>) | - <AFACTOR>

RT              ::= <TEXPR> {(<DESC> {,<DESC>...})
                  | (<DESC> {,<DESC>...})

TEXPR           ::= {<TEXPR> +} <TTERM> | <TEXPR> - <TTERM>

  Note: "+" is set union; "-" is set difference

TTERM           ::= {<TTERM> *} <TFACTOR>

  Note: "*" is set intersection
```

```
TFACTOR       ::= <RTNAME> | (<RT_COND>)

DESC          ::= <DESCNAME> | <RTNAME>.<DESCNAME>
                | <DESCNAME> OF <RTNAME>
                | <RTNAME>..<RTNAME>.<DESCNAME> | <FUNCTION>

FUNCTION      ::= <FUNCNAME> {(<DESC>{,<DESC>..})}

FUNCNAME      ::= AVG | MIN | MAX | SUM | COUNT | DOMAIN

RETRIEVE      ::= GET {RT} {<LABEL>:=} <RT_COND> {<ORDER>}
                | LET {RT} <LABEL>:= <RT_COND> {<ORDER>}

ORDER         ::= ORDER UP ON <DESC> {,<DESC>...}
                | ORDER DOWN ON <DESC> {,<DESC>...}

INDUCE        ::= INDUCE {<METHOD>} <LABEL>:=<DESCRIPTOR>
                      USING <KNOWLEDGE> {<PARMS>}
METHOD        ::= AQVAL | SYMMETRIC | UNICLASS | FEEDBACK

KNOWLEDGE     ::= <RTNAME>(<KTYPE>) {,<RTNAME>(<KTYPE>)...}
                |  <RTNAME> {,<RTNAME>...}

KTYPE         ::= RULE | CLASS=<CLASS>

CLASS         ::= <CLASS_NUMBER> | <DESCRIPTOR>

CLASS_NUMBER  ::= <INTEGER>

PARMS         ::= PARAMETERS(<PARM>:=<VALUE>
                          {,<PARM>:=<VALUE>...})

DEDUCE        ::= DEDUCE <DESCRIPTOR> FROM <RTNAME>
                          {WITH <EVENT>}

EVENT         ::= <EVENTNAME> | (<VALUE>{,<VALUE>...})

HELP          ::= HELP <LT> <NONTERMINAL> <GT>
                | HELP <INSTRUCTION>

INSTRUCTION   ::= DEFINE | ADD | GET | LET | CHANGE | DELETE
                | EXIT | COMMENT | INDUCE | DEDUCE | HELP
```

APPENDIX B

Meaning of the Elements of the Array PT

INSTRUCTION	ARRAY ELEMENT	MEANING
DEFINE	1	Length of PT
	2	1 (DEFINE)
	3	Length of PT used
	4	Number of RT's or Events defined
	For each RT defined	
	x	Length of PT used for this this RT
	x+1	1 (Indicates DEFINE RT)
	x+2	Index in SYMTABLE of the rtname
	x+3	Number of descriptors (nd)
	x+4	Number of key descriptors (nk)
	x+5	First descriptor number
	x+6	Second descriptor number
	.	.
	.	.
	.	.
	x+4+nd	Last descriptor number
	x+4+nd+1	First key descriptor number
	x+4+nd+2	Second key descriptor number
	.	.
	.	.
	.	.
	x+4+nd+nk	Last key descriptor number
	For each Event defined	
	x	Length of PT used for this Event
	x+1	2 (indicates DEFINE EVENT)
	x+2	Index in Symtable of event name
	x+3	Number of descriptors (nd)
	x+4	First descriptor number
	x+5	First value
	x+6	First value type
	x+7	Second descriptor number
	x+8	Second value
	x+9	Second value type
	.	.
	.	.
	.	.
	x+3nd+1	Last descriptor number
	x+3nd+2	Last value

	x+3nd+3	Last value type
ADD		
	1	Length of PT
	2	2 (ADD)
	3	Length of PT used
	4	Type of Add 1: ADD TO 2: ADD event 3: ADD (value, value,...)
	5	RT number to be added to
	6	Row condition operator 0: none 1: < 2: >
	7	Row number (or -1 if none given)
	For types 1 and 3	
	8	Number of events to be added (ne)
	9	Number of descriptors (nd)
	10	First value type 0: unknown ("?" entered) 1: integer 2: string or name 3: real
	11	Second value type
	.	.
	.	.
	.	.
	9+nd	Last value type code
	9+nd+1	First value for first event
	9+nd+2	Second value for first event
	.	.
	.	.
	.	.
	9+nd+ne	Last value for first event
	9+nd+ne+1	First value for second event
	.	.
	.	.
	.	.
	9+nd+nd*ne	Last value for last event
	For type 2	
	8	Event number
CHANGE		
	1	Length of PT
	2	3 (CHANGE)
	3	Length of PT used

```
        4               Number of tables to be changed
        5               First label
              First relational table condition
                        Second label
              Second relational table condition
                    .
                    .
                    .
                        Last label
              Last relational table condition

        For each CHANGE STATEMENT

        x               type of statement
                          1 Assignment
                          2 DISPLAY
                          3 GET
                          4 END
                          5 ABORT


DELETE
        1               Length of PT
        2               4 (DELETE)
        3               Length of PT used
              relational table condition


HELP
        1               LENGTH OF PT
        2               5 (HELP)
        3               Type of help
                          1: command
                          2: <nonterminal>
        4               Index in SYMTABLE of
                         command or nonterminal


GET and LET

        1               LENGTH OF PT
        2               6 (GET) or 7(LET)
        3               Length of PT used
        4               Index in SYMTABLE for label
              relational table condition
        x               Order operator
                          0: none
                          1: up
                          2: Down
        x+1             Number of descriptors to order
                          on (nd)
        x+2             First descriptor number
        x+3             Second descriptor number
```

.	.
.	.
.	.
x+1+nd	Last descriptor number

SAVE

1	Length of PT
2	8 (SAVE)
3	Length of PT used
4	Relational table number

The following is the format for a relational table condition

1	Number of relational tables in expression (nr)
2	Number of first relational table
3	Number of descriptors used in first table (nd)
4	First descriptor number
5	Second descriptor number
.	.
.	.
.	.
3+nd	Last descriptor number
3+nd+1	Number of descriptors used in 2nd table
.	.
.	.
.	.
x	Length of PT used for condition

Condition in Polish postfix

IL IOGRAPHIC DATA HET	1. Report No. UIUCDCS-R-77-846	2.	3. Recipient's Accession No.
tle and Subtitle 'he VL Relational Data Sublanguage for an Inferential 'omputer Consultant			5. Report Date October, 1977 6.
uthor(s) ichard N. Schubert			8. Performing Organization Rept. No.
rforming Organization Name and Address epartment of Computer Science niversity of Illinois at Urbana-Champaign rbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No. NSF MCS 74-03514
2. ponsoring Organization Name and Address lational Science Foundation ashington, D.C.			13. Type of Report & Period Covered 14.

5. upplementary Notes

6. Abstracts

An Inferential Computer Consultant is being designed and implemented at the niversity of Illinois by a research group headed by Professor R. S. Michalski. he computer consultant is intended to extend the capabilities of current informa- ion systems by including deductive capabilities of current information systems y including deductive capabilities and introducing inductive capabilities. In- uction is performed using Variable-Valued Logic techniques on sets of facts alled event sets. These event sets are most naturally stored using relational ables as proposed by Codd. In order to allow for the creation and manipulation f these relational tables, a data base sublanguage has been developed. The escription of this sublanguage is the object of this thesis.

7. ey Words and Document Analysis. 17a. Descriptors

ariable-Valued Logic	domain
nduction	tuple
eduction	key
ule	procedural
elational data base	PASCAL
elation	interactive computer program
etrieval	hard coded parser
vent	recursive descent parsing
escriptor	postfix string

7b. Identifiers/Open-Ended Terms

7c. COSATI Field/Group

8. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
	20. Security Class (This Page UNCLASSIFIED	22. Price